View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Juelich Shared Electronic Resources

John von Neumann Institute for Computing

Parallel Jacobian Accumulation

Ebadollah Varnik, Uwe Naumann

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 311-318, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

Parallel Jacobian Accumulation

Ebadollah Varnik and Uwe Naumann

Department of Computer Science
RWTH Aachen University, D-52056 Aachen, Germany
E-mail: {varnik, naumann}@stce.rwth-aachen.de

The accumulation of the Jacobian matrix F' of a vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ implemented as a computer program can be regarded as a transformation of its linearized computational graph into a subgraph of the directed complete bipartite graph $K_{n,m}$. This transformation can be performed by applying a vertex elimination technique. We report on first results of a parallel approach to Jacobian accumulation.

1 Introduction

In *automatic differentiation*[1–3] we consider implementations of vector functions

$$F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x}) \quad ,$$

as computer programs written in some imperative programming language, that map a vector $\mathbf{x} \equiv (x_i)_{i=1,\ldots,n}$ of *independent* variables onto a vector $\mathbf{y} \equiv (y_j)_{j=1,\ldots,m}$ of *dependent* variables. We assume that F has been implemented as a computer program. Following the notation in Griewank's book[4] we assume that F can be decomposed into a sequence of p single assignments of the value of scalar *elemental* functions φ_i to unique *intermediate* variables v_j. The *code list* of F is given as

$$(\mathbb{R} \ni)\ v_j = \varphi_j(v_i)_{i \prec j} \quad , \tag{1.1}$$

where $j = n+1, \ldots, q$ and $q = n + p + m$. The binary relation $i \prec j$ denotes a direct dependence of v_j on v_i. The variables $\mathbf{v} = (v_i)_{i=1,\ldots,q}$ are partitioned into the sets X containing the *independent* variables $(\mathbf{v}_i)_{i=1,\ldots,n}$, Y containing the *dependent* variables $(v_i)_{i=n+p+1,\ldots,q}$, and Z containing the intermediate variables $(v_i)_{i=n+1,\ldots,n+p}$. The code list of F can be represented as a directed acyclic *computational graph* $G = G(F) = (V, E)$ with integer vertices $V = \{i : i \in \{1, \ldots, q\}\}$ and edges $(i, j) \in E$ if and only if $i \prec j$. Moreover, $V = X \cup Z \cup Y$, where $X = \{1, \ldots, n\}$, $Z = \{n+1, \ldots, n+p\}$, and $Y = \{n+p+1, \ldots, q\}$. Hence, X, Y, and Z are mutually disjoint. We distinguish between *independent* ($i \in X$), *intermediate* ($i \in Z$), and *dependent* ($i \in Y$) vertices. Under the assumption that all elemental functions are continuously differentiable in some neighbourhood of their arguments all edges (i, j) can be labeled with the partial derivatives $c_{j,i} \equiv \frac{\partial v_j}{\partial v_i}$ of v_j w.r.t. v_i. This labelling yields the *linearized* computational graph G of F. Eq. (1.1) can be written as a system of nonlinear equation $C(\mathbf{v})$ as follows[4].

$$\varphi_j(v_i)_{i \prec j} - v_j = 0 \quad \text{for } j = n+1, \ldots, q \quad .$$

Differentiation with respect to $\mathbf{v}$ leads to

$$C' = C'(\mathbf{v}) \equiv (c'_{j,i})_{i,j=1,\ldots,q} = \begin{cases} c_{j,i} & \text{if } i \prec j \\ -1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \quad .$$

The *extended Jacobian* C' is lower triangular. Its rows and columns are enumerated as $j, i = 1, \ldots, q$. Row j of C' corresponds to vertex j of G and contains the partial derivatives $c_{j,k}$ of vertex j w.r.t. all of its predecessors. In the following we refer to a row i as *independent* for $i \in \{1, \ldots, n\}$, as *intermediate* for $i \in \{n+1, \ldots, n+p\}$, and as *dependent* if $i \in \{n+p+1, \ldots, q\}$.

The Jacobian matrix $F' = F'(\mathbf{x}) = \left(\frac{\partial y_j}{\partial x_i}(\mathbf{x})\right)_{i=1,\ldots,n}^{j=1,\ldots,m} \in I\!R^{m \times n}$ of F at point $\mathbf{x}$ can be computed on the linearized computational graph by elimination[5,6] of all intermediates vertices as introduced in Griewank et al.[7] resulting in a bipartite graph $G' = (\{X, \emptyset, Y\}, E')$ with labels on the edges in E' representing exactly the nonzero entries of F'. Following the chain rule an intermediate vertex $j \in Z$ can be eliminated by multiplying the edge labels over all paths connecting pairwise the predecessors i and successors k followed by adding these products. The result is the label of the edge (i, j).

The elimination of intermediate vertex j can be also understood as elimination of all non-zero entries in row/column j of C'[8]. Therefore one has to find all those rows k with $j \prec k$ and eliminate this dependency according to the chain rule. Our implementation is based on a *Compressed Row Storage* (CRS)[9] representation of C'. However we believe that an explanation in terms of the computational graph will be easier to follow. References to the CRS representation are included where necessary or useful.

Example: Consider a vector function $f : I\!R^2 \to I\!R^2$ with the code list given in Fig. 1 (a). The corresponding G and C' are shown in Fig. 1 (b) and (c), respectively. The vertices [rows] 1 and 2 represent independent, 6 and 7 dependent, and 3, 4, and 5 intermediate vertices [rows]. Consider row 4 in Fig. 1 (c) containing the partials $c_{5,3}$ and $c_{5,4}$. These are labels of incoming edges $(3, 5)$ and $(4, 5)$ of vertex 5 in Fig. 1 (b). Column 5 contains the partial derivatives $c_{6,5}$ and $c_{7,5}$ that are the labels of the outgoing edges $(5, 6)$ and $(5, 7)$ of vertex 5.

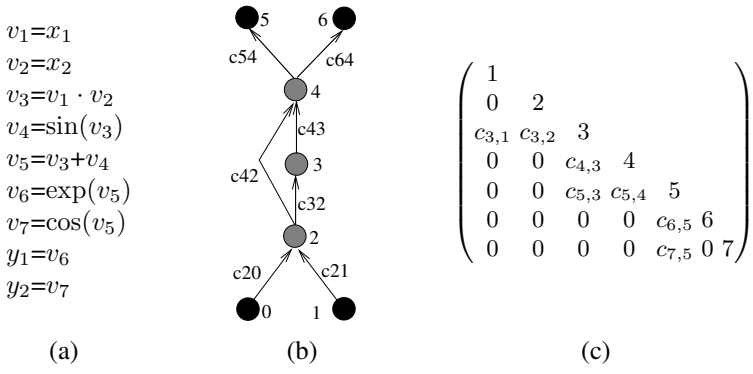
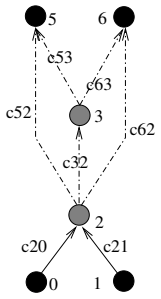


Figure 1. Code list (a), linearized computational graph G (b) and extended Jacobian C' (c) of f. The diagonal entries of C' mark the row index.

Eliminating $c_{6,5}$ in C' as shown in Fig. 2 (b) is equivalent to *back-elimination*[10] of $(5, 6)$ as shown in Fig. 2 (a). *Fill-in* is generated as $c_{6,3}$ $[(3, 6)]$ and $c_{6,4}$ $[(4, 6)]$ since row

[vertex] 6 has a non-zero [incoming edge] in [from] column [vertex] 5. The elimination of row [vertex] 5 on C' [G] can be considered as elimination [back-elimination] of all partial derivatives [outedges] $c_{k,5}$ with $5 \prec k$ [$(5,k)$]. Further elimination of intermediate rows [vertices] 4 and 3 on C' [G] as shown in Fig. 3 (b) [(a)] yield the Jacobian entries in [as] the first two column's [labels of the incoming edges] of the dependent rows [vertices] 6 and 7.



(a)

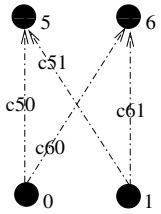
$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ c_{3,1} & c_{3,2} & 3 & & & \\ 0 & 0 & c_{4,2} & 4 & & \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 5 & \\ 0 & 0 & \mathbf{c_{6,3}} & \mathbf{c_{6,4}} & \mathbf{0} & 6 \\ 0 & 0 & \mathbf{c_{7,3}} & \mathbf{c_{7,4}} & \mathbf{0} & 0 \; 7 \end{pmatrix}$$

(b)

$$\begin{aligned} c_{6,3} &= c_{6,5} \cdot c_{5,3} \\ c_{6,4} &= c_{6,5} \cdot c_{5,4} \\ c_{7,3} &= c_{7,5} \cdot c_{5,3} \\ c_{7,4} &= c_{7,5} \cdot c_{5,4} \end{aligned}$$

(c)

Figure 2. Computational graph G (a) and extended Jacobian C' (b) of f after elimination of vertex and row 5, respectively.



(a)

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ \mathbf{0} & \mathbf{0} & 3 & & & \\ 0 & 0 & \mathbf{0} & 4 & & \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 5 & \\ \mathbf{c_{6,1}} & \mathbf{c_{6,2}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 6 \\ \mathbf{c_{7,1}} & \mathbf{c_{7,2}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \; 7 \end{pmatrix}$$

(b)

$$\begin{aligned} c_{6,3} &+= c_{6,4} \cdot c_{4,3} \\ c_{7,3} &+= c_{7,4} \cdot c_{4,3} \\ c_{6,1} &= c_{6,3} \cdot c_{3,1} \\ c_{6,2} &= c_{6,3} \cdot c_{3,2} \\ c_{7,1} &= c_{7,3} \cdot c_{3,1} \\ c_{7,2} &= c_{7,3} \cdot c_{3,2} \end{aligned}$$

(c)

Figure 3. Bipartite graph G (a) and extended Jacobian C' after elimination of all intermediate vertices and rows, respectively.

2 Parallel Algorithms

In order to parallelize the Jacobian accumulation by vertex elimination, we decompose the computational graph G of F into k subgraphs $G_p = (V_p, E_p)$ with $p \in \{1 \cdots k\}$, $V_p \in V$ and $E_p = \{(i,j) \in E | v_i, v_j \in V_p\}$, where $V = \cup_{p=1}^{k} V_p$ and $E = \cup_{p=1}^{k} E_p$ with $E_p \cap E_q = \emptyset$ for $p, q \in \{1, \cdots, k\}$. Moreover, $V_p = X_p \cup Y_p \cup Z_p$, with vertex cuts X_p and Y_p representing the *local independent* and *local dependent* vertices of the subgraph G_p, respectively. Z_p represents the set of *local intermediate* vertices k, which lie on a path from a vertex $i \in X_p$ to a vertex $j \in Y_p$, with $k \in V_p - \{X_p \cup Y_p\}$. Hence, X_p,

Y_p, and Z_p are mutually disjoint. We call subgraphs G_i and G_j neighbors, if Y_i=X_j with $i, j \in 1, k-1$ and this is only the case, if $j = i + 1$. Whenever we talk about *interface* vertices, we mean the common vertices of two neighbors. We refer to subgraphs G_p as *atomic* subgraphs in the sense that all edges $(i, j) \in E_p$ are between vertices $i, j \in V_p$ of the same subgraph.

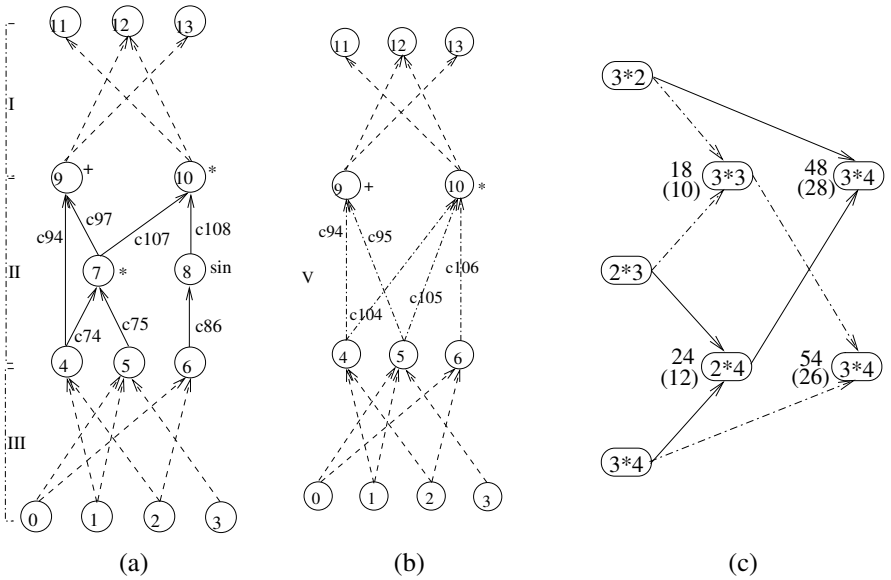


Figure 4. Graph decomposition (a), local Jacobians by vertex elimination (b), optimal matrix product (c).

In this step the main focus is on having balanced subgraphs to optimize the computational and communication effort in concurrent processes. Applying the vertex elimination technique to a subgraph G_p yields the local Jacobian matrix F'_p. Considering the subgraph G_2 in Fig. 4 (a) the elimination of vertices 8 and 9 yields the corresponding bipartite graph G'_2 as shown in Fig. 4 (b). The edge labels of G'_2 correspond to the entries of the local Jacobian

$$F'_2 = \begin{pmatrix} c_{10,5} & c_{10,6} & 0 \\ c_{11,5} & c_{11,6} & c_{11,7} \end{pmatrix} \quad .$$

The reduction to the Jacobian matrix $F' = \prod_{p=1}^{k} F'_p$ can be considered as the chained product of k local Jacobian matrices. For the decomposed computational graph in Fig. 4 (b) with $k = 3$ the Jacobian $F' = {F'_3}^{3*2} \times {F'_2}^{2*3} \times {F'_1}^{3*4}$ is the chained product of local Jacobians F'_3,F'_2, and F'_1. Dynamic programming can be used to optimize the number of floating point multiplications (FLOPS) arising in the chained matrix product as shown in Fig. 4 (c), and described in Griewank et al.[11].

Our implementation uses OpenMP[12,13]. It runs on the shared memory system Solaris Sun Fire E6900 with 16 Ultra Sparc VI 1.2 GHz Dual Core processors and 96 GByte

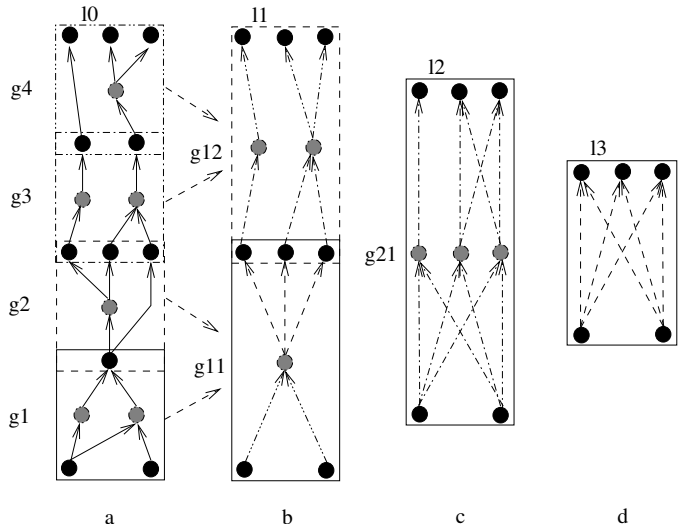


Figure 5. Parallel Jacobian accumulation by vertex elimination using Pyramid strategy.

Memory. In the following we present two ideas for parallelizing the process of Jacobian accumulation by vertex elimination on the computational graph.

2.1 Pyramid Approach

The pyramid approach realizes a level-based parallel vertex elimination illustrated with an example in Fig. 5. At the lowest level $l = 0$ (Fig. 5 (a)) the computational graph G consists of 4 atomic subgraphs. The gray coloured vertices represent the local intermediate vertices of subgraphs. For example the subgraph G_3^0 has 2 local intermediate, 3 local independent, and 2 local dependent vertices, respectively. Applying the vertex elimination on all 4 subgraphs in parallel yields the computational graph as shown in Fig. 5 (b) at level $l = 1$. G gets decomposed into 2 subgraphs, namely G_1^1 and G_2^1. The decomposition at level $l > 0$ is nothing else than merging two neighboring subgraphs into one. The interface vertices of two neighbors become local intermediates of the current subgraph. For instance G_1^1 Fig. 5 (b) at level $l = 1$ results from merging G_1^0 and G_2^0 after elimination of their local intermediate vertices. Repeating this process for G_1^1 and G_2^1 yields G^2 as shown in Fig. 5 (c). After elimination of 3 local intermediates of G_1^2 we get the bipartite graph G_3 shown in Fig. 5 (d). It is obvious that the elimination process at level $l = 2$ proceeds serially.

2.2 Master-Slave Approach

The master-slave approach[14] consists of two steps: *elimination* and *reduction*. The former is the same as the vertex elimination on subgraph G_p yielding the local bipartite graph that corresponds to local Jacobian F'_p. This step is performed by the slaves in parallel. The latter multiplies the local Jacobians and is performed by the master. In graphical terms the

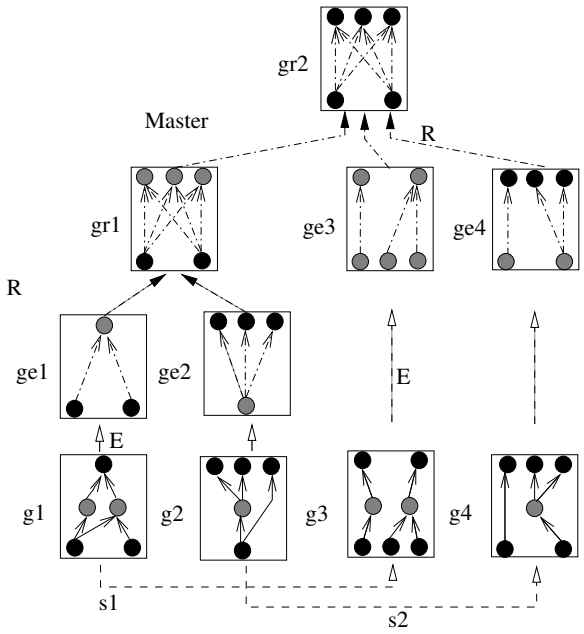


Figure 6. Parallel Jacobian accumulation by vertex elimination using Master-Slave strategy.

reduction step can be understood as the elimination of interface vertices of two neighboring local bipartite graphs. The underlying computational graph and its decomposition as shown in Fig. 6 are the same as in Fig. 5 (a). The example in Fig. 6 illustrates the master-slave idea using 2 slaves s_1 and s_2. The slaves s_1 and s_2 apply vertex elimination to the subgraph G_1 and G_1, respectively. Each slave, for instance s_1, gets the next job (subgraph) G_3 immediately after termination of the previous job. The local bipartite graphs G_1^e and G_2^e shown in Fig. 6 are the result of previous elimination. They are reduced by the master to G_1^r.

Conclusion and Numerical Results

Our implementation consists of two main steps: *symbolic* and *accumulation*. The symbolic step proceeds on a bit pattern[8] and detects the fill-in generated during the accumulation process on CRS. Different elimination techniques can yield different fill-in schemes, which have a big impact on both memory and runtime requirement of the Jacobian accumulation. Fig. 7 presents first numerical results of parallel Jacobian accumulation on CRS of the extended Jacobian of a $2D$ discretization of the Laplace equation comparing the pyramid (PYRAMID) and master-slave (MASTERSLAVE) approaches with the serial (SERIAL) version. The serial version computes the Jacobian by applying reverse elimination to the entire CRS of the underlying problem. Using two threads our parallel approaches are roughly three times faster than the serial version for large problem sizes. Partly this speedup is caused by the difference in the generated fill-in as shown in Fig. 8,

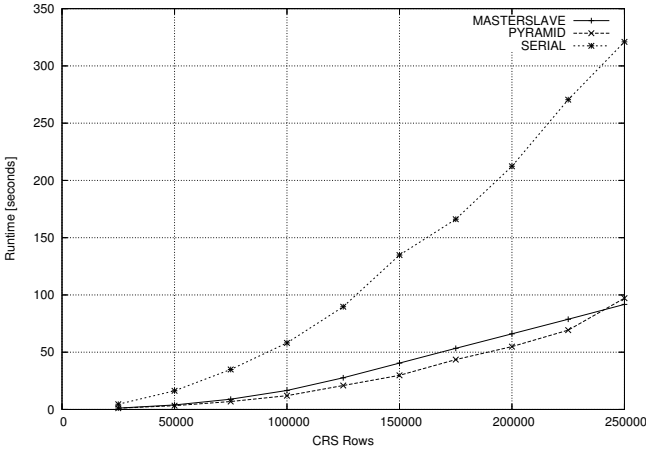


Figure 7. Runtime analysis of parallel Jacobian accumulation on CRS in reverse order against the serial version.

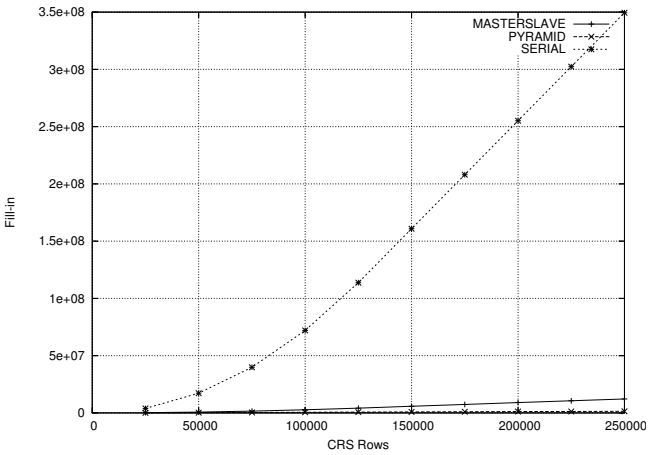


Figure 8. Fill-in comparison between SERIAL, MASTERSLAVE, and PYRAMID approaches.

which has a big impact on the runtime and memory requirement of the elimination process. The real runtime contribution of the fill-in is the subject of ongoing research. Our first results show that for parallelizing the Jacobian accumulation on CRS fill-in has to be taken into account, which makes the parallelization an even more complex task. Further work will focus on optimizing both fill-in generation and parallelization approaches thus aiming for better memory and runtime behaviour.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, (Eds.), *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, (SIAM, 1996).
2. G. Corliss and A. Griewank, (Eds.), *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, (SIAM, 1991).
3. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, (Eds.), *Automatic Differentiation of Algorithms – From Simulation to Optimization*, (Springer, New York, 2002).
4. A. Griewank, *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, Number 19 in Frontiers in Applied Mathematics, (SIAM, Philadelphia, 2000).
5. A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in: Automatic Differentiation of Algorithms: Theory, Implementation, and Application, pp. 126–135, (SIAM, Philadelphia, PA, 1991).
6. U. Naumann, *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph*, Math. Prog., **99**, 399–421, (2004).
7. A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markovitz rule*, in: Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, (Eds.), pp. 126–135, (1991).
8. E. Varnik, U. Naumann, and A. Lyons, *Toward low static memory Jacobian accumulation*, WSEAS Transactions on Mathematics, **5**, 909–917, (2006).
9. I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices*, (Clarendon Press, Oxford, 1986).
10. U. Naumann, *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*, PhD thesis, Technical University of Dresden, (1999).
11. A. Griewank and U. Naumann, *Accumulating Jacobians as chained sparse matrix products*, Math. Prog., **3**, 555–571, (2003).
12. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*, (Morgan Kaufmann, San Francisco, 2001).
13. M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, (McGraw-Hill Education, ISE Editions, 2003).
14. C. H. Bischof, H. M. Bücker, and P.T. Wu, *Time-parallel computation of pseudo-adjoints for a leapfrog scheme*, International Journal of High Speed Computing, **12**, 1–27, (2004).